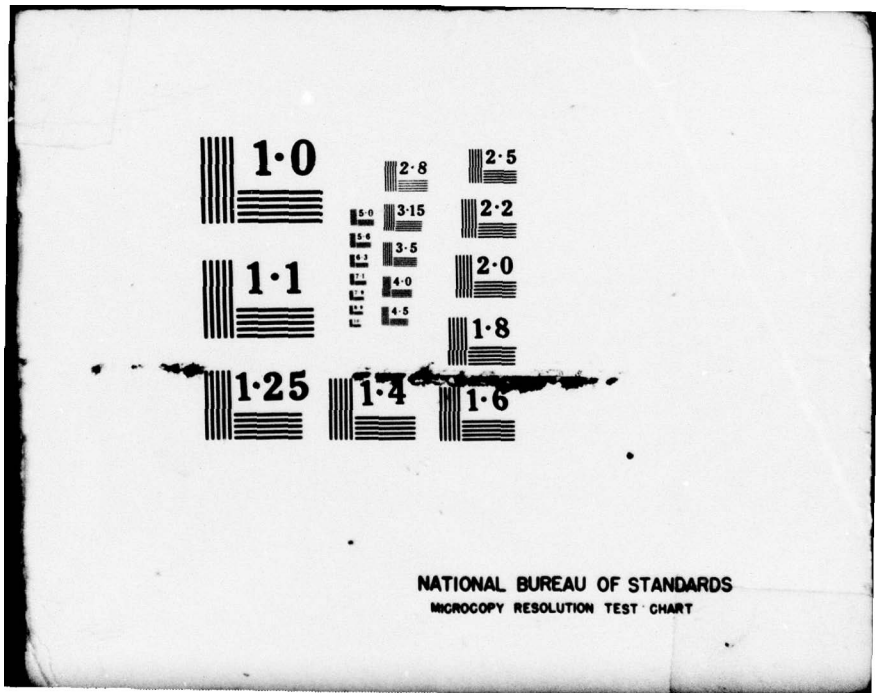
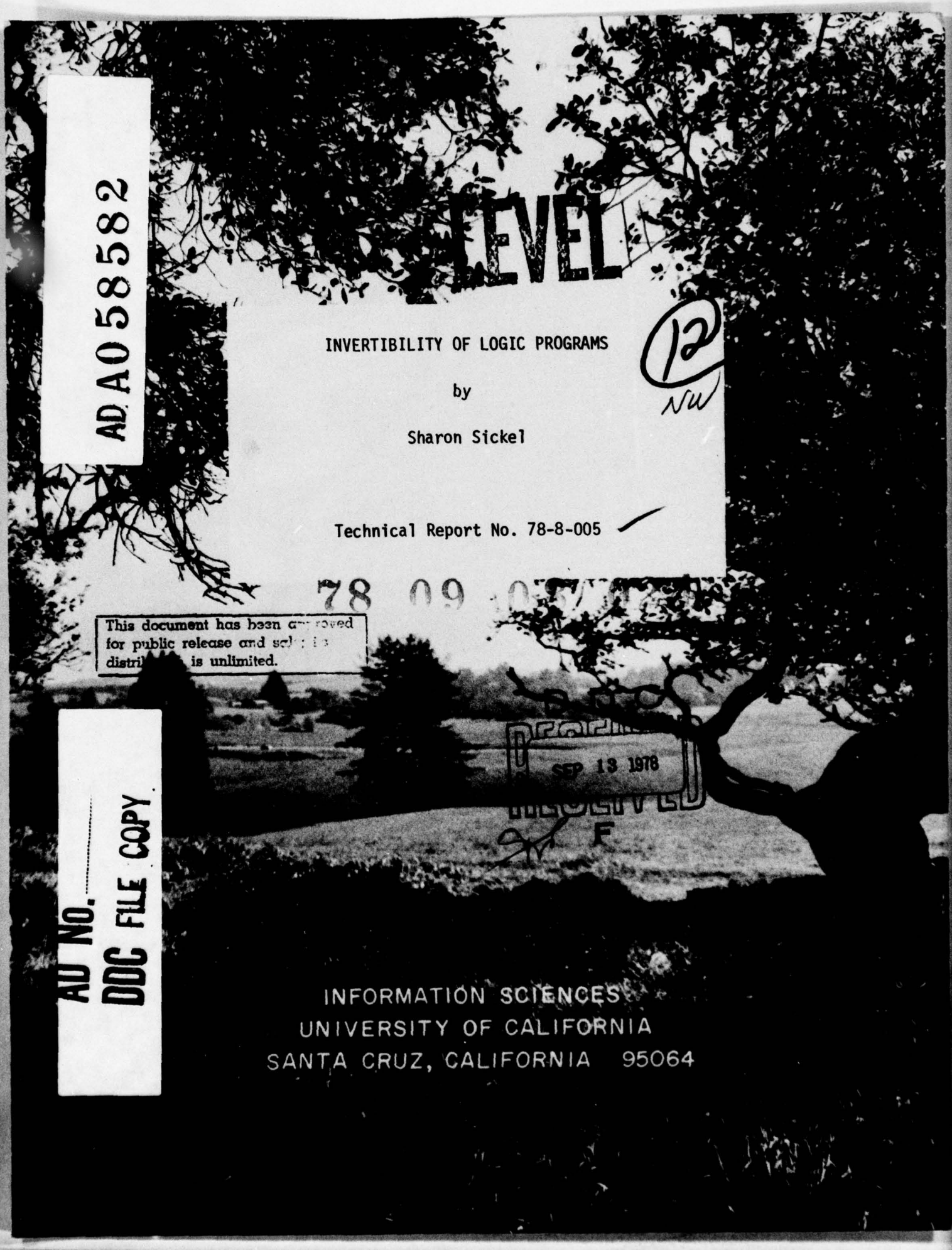
AD-A058 582 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
INVERTIBILITY OF LOGIC PROGRAMS.(U)
AUG 78 S SICKEL N00014-76-C-0682
UNCLASSIFIED TR-78-8-005 NL

| OF |
ADA
058582

END
DATE
FILMED
11-78
DDC


1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
5·0
2·5
2·2
2·0
1·8
1·4
1·6
NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART



LEVEL

(12)
NW

INVERTIBILITY OF LOGIC PROGRAMS

by

Sharon Sickel

Technical Report No. 78-8-005

INFORMATION SCIENCES
UNIVERSITY OF CALIFORNIA
SANTA CRUZ, CALIFORNIA 95064

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
(6) INVERTIBILITY OF LOGIC PROGRAMS.

5. TYPE OF REPORT & PERIOD COVERED
(9) Technical Repts

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
(10) Sharon Sickel

8. CONTRACT OR GRANT NUMBER(s)
(15) N00014-76-C-0682

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Information Sciences
University of California
Santa Cruz, California 95064

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
(11) 1 Aug 78

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research
Arlington, Virginia 22217

12. REPORT DATE
August 1, 1978

13. NUMBER OF PAGES
17

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Office of Naval Research
University of California
553 Evans Hall
Berkeley, California 94720

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

(12) 22 p.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

(14) TR-78-8-005

DDC RECEIVED SEP 13 1978 F

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number) and Phrases:

Logic programming, relational programming, invertible functions, recursion, analysis of programs.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Predicates can describe functions; the arguments of the predicates are the input and output parameters of the function. Logic programs describe relationships between objects rather than merely sequential instructions, and it is common for both a function and its inverse to be computable by the same logic program [1]. Given values for some subset of the arguments to a function-describing predicate, one may be able to decide, in general, which of the remaining arguments are computable by the logic program.

(Continued on next page)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102 LF 014 6601

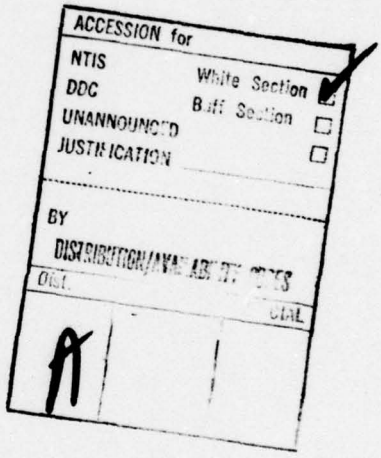
410350

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

78 09 08 024

20. (Continued)

The concept of functional inverse can be generalized in the context of logic programming. A new kind of inverse, called j-inverse, is defined. Two algorithms which analyze and test the recursive structure of logic programs for any specific invertibility are presented. A set of guidelines to help the logic programmer construct j-invertible programs is given.



INVERTIBILITY OF LOGIC PROGRAMS

by

Sharon Sickel

Information Sciences

and

Crown College

University of California

Santa Cruz

This work was supported by Office of Naval Research Contract #76-C-0682.

ABSTRACT

Predicates can describe functions; the arguments of the predicate are the input and output parameters of the function. Logic programs describe relationships between objects rather than merely sequential instructions, and it is common for both a function and its inverse to be computable by the same logic program [1]. Given values for some subset of the arguments to a function-describing predicate, we may be able to decide, in general, which of the remaining arguments are computable by the logic program.

The concept of functional inverse can be generalized in the context of logic programming. A new kind of inverse, called j-inverse, is defined. Two algorithms which analyze and test the recursive structure of logic programs for any specific invertibility are presented. A set of guidelines to help the logic programmer construct j-invertible programs is given.

1. Introduction

A function is a mapping $f:D \rightarrow R$ or $f(x) = y$. $f(x)$ must have a unique value, that is to say it must map to exactly one value in R for any given element of D. We can extend that definition by allowing both the input and the output to be tuples. For example, $f:D_1 \times D_2 \times \ldots D_n \rightarrow R_1 \times R_2 \times \ldots R_k$ or $f(x_1,\ldots,x_n) = (y_1,\ldots,y_k)$. We say that f is invertible if there exists a function f^{-1} such that $f^{-1}(y) = x$ iff $f(x) = y$.

We wish to restrict the notion of functional inverse as follows:

If $f(x_1,\ldots,x_n) = y$, then we say that f is <u>j-invertible</u> if f_j exists as follows:

$$f_1(x_2,\ldots,x_n,y) = x_1$$
$$f_2(x_1,x_3,\ldots,x_n,y) = x_2$$
$$\vdots$$
$$f_n(x_1,\ldots,x_{n-1},y) = x_n$$

That is to say, given any n values of the n+1 tuple $(x_1,\ldots,x_n,y)$, we can find the remaining one. f_j is the j-inverse of f. The value y maybe a k-tuple but for the purposes here we do not decompose it as we do the input n-tuple.

If a function f is invertible, then it is j-invertible for all j, $1 \leq j \leq n$, since invertibility implies that given y we can find $x_1,\ldots,x_n$, so any values x_i that are provided are redundant. The opposite is not true however. Given a function f, even if it is j-invertible for all $1 \leq j \leq n$, f^{-1} does not necessarily exist. A counterexample is the arithmetic function, add, on integers:

$$add(a,b) = c$$

There are, in general, many pairs (a,b) that add maps to c; so there is no way to define add^{-1}. However, add is 1- and 2-invertible; $add_1(a,c) = c-a$ and $add_2(b,c) = c-b$.

A function is invertible, in general, when the functional mapping is 1-1 and onto. A function is j-invertible if and only if all pairs of (n+1)-tuples $(x_1,\ldots,x_n,y)$ such that $f(x_1,\ldots,x_n) = y$ differ at position j only if they differ in at least one

more position. There are many interesting functions that are not invertible in general but that are j-invertible for some j. Given a function, we frequently would like to be able to define a single program to compute this function that can also compute the partial inverses.

2. Invertibility of Functions Expressed as Logic Programs.

A logic program [3] is a set of WFs in the form L ← R and one WF, the call, of the form ← R where L is a predicate and R is a conjunction of predicates. All variables are implicitly universally quantified. Logic programs are theorems describing relationships among objects. While an interpretation of these logic theorems can drive a computation, there is no notion of certain parameters for input and others for output. A nontrivial invertible logic program is given in another paper [2].

We now define some terms that will be needed later. A logic procedure is an individual implication in logic program, A ← B where B may be empty; A is the procedure head; B is the procedure body. A termination condition of a recursion is a procedure A ← B such that B does not contain a predicate whose name is the same as the predicate name of A. That is to say, it contains no recursive calls to the procedure. We say that a variable, x, drives the computation of a procedure, if every recursion causes the value of x to be nearer a value that will cause the recursion to terminate.

We can write a logic program to compute

$$exp(x,y) = x^y = z$$

by successive multiplications. We assume a 2-invertible predicate MULT(x,y,z) which is true if and only if $x \cdot y = z$. This exponential function is defined by predicate EXP1, which has the semantics that EXP1(x,y,z) is true if and only if $x^y = z$.

The logic program is:

$$EXP1(x+1,0,1) \leftarrow \S$$

$$EXP1(x,y+1,w) \leftarrow EXP1(x,y,z) \wedge MULT(x,z,w)$$

The meaning of A ← B is the standard logic interpretation "A is implied by B". The meaning of the program is

$$(x+1)^0 = 1$$

$$(x^{y+1} = x \cdot z) \leftarrow (x^y = z)$$

It is computationally effective as can be seen by the example below. Starting with the call, we get a terminating sequence of sub-goals:

$$\leftarrow EXP1(3,2,answer)$$

$$EXP1(3,1,z) \wedge MULT(3,z,answer)$$

$$EXP1(3,0,z') \wedge MULT(3,z',z)$$

The last call to EXP1 gives z' = 1 by the termination condition, MULT then gives z = 3 and then answer = 9 or EXP1(3,2,9), as desired.

Suppose we wish to 2-invert EXP1. For example, evaluate $3^Y = 9$ to get Y = 2. The sequence of calls starts

$$\leftarrow EXP1(3,Y,9)$$

$$EXP1(3,y,z) \wedge MULT(3,z,9)$$

Since MULT is 2-invertible, we get z = 3; hence substituting 3 for z we derive EXP1(3,y,3) where Y is bound to y+1. Continuing we get new subgoals

$$EXP1(3,y',z') \wedge MULT(3,z',3)$$

with y bound to y'+1. MULT(3,z',3) gives z' = 1. Then in evaluating EXP1(3,y',1)

---

§ In EXP(x+1,0,1) ← the expression x+1 is used to denote a nonzero value. Another representation for this is

EXP(x,0,1) ← (x > 0), but the latter has a slight computational disadvantage.

the recursive case fails because there is no z" such that MULT(3,z",1). However, the terminating condition applies; thus, y' = 0, y = 1, Y = 2, and we derive EXP1(3,2,9), as desired.

Suppose we wish to 1-invert EXP1. For example, for what value of X will $X^3 = 27$? The following sequence of subgoals is generated:

$$\leftarrow \text{EXP1}(X,3,27)$$

$$\text{EXP1}(X,2,w) \wedge \text{MULT}(X,w,27)$$

$$\text{EXP1}(X,1,w') \wedge \text{MULT}(X,w',w)$$

$$\text{EXP1}(X,0,w'') \wedge \text{MULT}(X,w'',w')$$

The termination condition gives us w" = 1 and the definition of MULT yields X = w' giving

$$\text{EXP1}(X,1,X) \wedge \text{MULT}(X,X,w)$$

But we fail here because while MULT can, awkwardly, compute square root by calling MULT(x,x,C) where C is a given constant natural number, it cannot compute square roots where the third argument is unknown as well. In effect, the collective calls to MULT are taking n-th roots of the original third argument, and that is beyond the capability of MULT.

Now we make a distinction between mathematical invertibility and computational invertibility. A function f may be mathematically invertible but be specified as a logic program that will not compute the inverse. If a function is computationally invertible it is necessarily mathematically invertible.

It may also be the case that a mathematically correct logic program will be computationally reasonable in one direction and horribly inefficient in the other. The path of choices in a computation may be forking such that in one direction many cases are being joined together as in Figure 1a, and in the other direction a choice must be made from many alternatives as in Figure 1b. The latter case may require much

backtracking where incorrect choices are made.

(a) (b)

Figure 1

3. When are Logic Programs Invertible?

The process of interpreting a logic program starts at the call, decends recursively to a termination condition, then returns eventually to the call. The question of j-invertibility depends upon the sequence of bindings of variables as the recursion is carried out; success is achieved when the (unbound) j-th variable of the call is bound to a constant value. More generally, if any subset of the variables is known at the call, we can ask if all unknown values can be computed.

Parameter list patterns are used to designate in a parameter list which parameters are known (i.e. constants) and which are unknown (i.e. variables or functions of variables). Given a parameter list input to a procedure, we can construct an input pattern or input template by replacing constants in the parameter list by 1's, and replacing variables or functions[§] of variables by 0's. So, for example, from parameter list (2,x,y+1) we construct input template (1,0,0). We may tie parameters together if some of the unknown variables are the same or simple functions of each other, such that if one can be computed, the other is known as well. E.g. parameter list (2,x,x+1) corresponds to pattern $(1,\overset{\frown}{0,0})$.

We have two algorithms that map input templates, i.e. known vs. unknown input

---

[§] We assume the only functions that are allowed as part of the parameter lists are simple constructor functions, e.g. +1.

parameters, to output templates that show what total set of values is known after the computation. For example, for procedure MULT, input pattern (101) maps to (111), since given one factor and the product, the other factor can be determined.

The first algorithm is a simple, easy to compute, function that gives a strong indicator of the pattern mapping, but pathological cases can be constructed in which the computed mapping will be stronger than reality. The second algorithm is similar to the first, but makes added assumptions that require some meta-analysis of the given function. The second algorithm is a sufficient test, i.e. at least all of the values claimed to be known, will be known.

An input pattern tells what input values are known. From that we can show where known values appear in the entire procedure. For example, input pattern (101) given to the recursive case of EXP1 gives procedure pattern:

$$\text{EXP1}(101) \leftarrow \text{EXP1}(100) \wedge \text{MULT}(101)$$

or (101) (100) (101) in abbreviated form.

The procedure operates in two stages:

1) Going down the recursion, we check off values when we know that they are computable. Even though the recursion can go arbitrarily deep, there are a bounded number of procedure patterns.

2) Once the derivable procedure patterns are found, patterns of the termination conditions are applied to show which values will be given by termination.

Throughout both stages, propagate new values for variables wherever the variables appear, and when applicable, replace the input patterns of sub-functions by the output patterns to which they map.

Weak-Heuristic Algorithm:

Purpose: To discover the input-to-output mappings of a predicate, Q.

Hypotheses: All auxiliary functions' input-to-output mappings are given. The only functions appearing in terms are constructor functions.

1. Set-up.

The input pattern is determined from the call: ← Q(p1, ..., pn). We replace in the call:

a) each constant or constructor applied to a constant by the value 1.

b) each constructor applied to a variable by the variable itself.

The procedure templates are determined from logic procedures of the form

$$Q(t1, \ldots, tn) \leftarrow R1 \wedge \ldots \wedge Rk.$$

We replace each term in the procedure as in 1a and 1b, above.

Let PPS be the set containing the input pattern.

2. Propagation of known values.

While there is a conjunct Q(p1, ..., pn) appearing in the body of a procedure pattern of PPS such that Q(p1, ..., pn) does not appear as the head of an element of PPS, create new elements of PPS from the procedure templates as follows:

a) Unify the head of the template with Q(p1, ..., pn).

b) The auxiliary functions are the subgoals of the procedure template that are not recursive calls to Q. Unify the inputs to the auxiliary functions with their corresponding output mapping. (This records which variables become

bound in the course of the computation of the auxiliary function.) This step is repeated until no more changes are possible.

c) Add this new procedure pattern to PPS.

3. Applying termination cases.

a) Partition the elements of PPS into recursive and non-recursive.

b) Select a recursive subgoal from the body of an element of PPS, and unify it with the head of any non-recursive element. Apply the bindings only to the recursive pattern. Drop the recursive subgoal that was just unified. The modified recursive procedure may now be non-recursive. If so, move it to the non-recursive partition.

c) Repeat b) until the recursive partition is empty.

4. Interpretation of results.

The output pattern Q(p1, ..., pn) such that (p1, ..., pn) is the most specific parameter list that will unify with the parameter lists of all the heads of the procedures of PPS.

Strong Heuristic Algorithm:

This algorithm is the same as the previous one except that where 1's are unified with 1's, a case-by-case meta-argument is required to show that the actual constants are unifiable. If such is not the case, the procedure pattern being generated is discarded.

For example, check to see if EXP1 is 2-invertible, i.e. whether input template

(101) maps to output template (111).

Procedure template is	:	(101) (100) (101)
MULT maps (101) to (111):		(101) (100) (111)
Fill in new value of z wherever it appears	:	(101) (101) (111)
Termination condition fills in y	:	(101) (111) (111)
Fill in new value of y wherever it appears	:	(111) (111) (111)
Output template is the result in procedure head:		(111)

So, the original call has gone from (101) to (111), suggesting 2-invertibility.

The preceding algorithms map input templates to output templates, and in particular check whether a logic program is j-invertible. The following are a set of guidelines that help the logic programmer to construct j-invertible programs. These guidelines exclude some j-invertible logic programs and are thus overly restrictive. More liberal, but less intuitive guidelines, can be derived from the preceding algorithm.

A. Termination conditions: There should be a set of termination conditions, one of which can always be reached from a call in which the j-th parameter is the only unknown, such that each element in the set has the property that the j-th parameter is known or can be directly computed as a function of some of the other arguments.

For example, consider the termination condition MULT(x,0,0) ←. It is acceptable for 2-invertibility because the second parameter is a constant and we always reach this termination condition. However, it is not acceptable for 1-invertibility since once we recur and bottom-out, we still have no way to establish the value of x except by evaluating algebraic formulas. For example, MULT(x,2,6) calls MULT(x,1,z) where z = 6-x which calls MULT(x,0,z') where z' = 6-x-x which = 0. So with a little algebra

we can deduce that x = 3. However, this is not what is meant by directly computable.

B. Invertible subfunctions: The other functions used in the definition must be invertible as called.

For example, consider the following predicate:

$$F(x_1,x_2+1,y) \leftarrow F(x_1,x_2,y') \wedge G(y',y)$$

For F to be 1-invertible, G must be 1-invertible.

C. Driving the Computation: At least one known value must drive the computation. For example, consider the function F, above. If x_2 is unknown, it cannot drive the computation. So for F to be 2-invertible, since x_1 does not change, the mapping $y \underset{G}{\rightarrow} y'$ must drive toward a termination condition.

A way to guarantee that this property holds for all j, $1 \leq j \leq n$ is to have at lease two arguments driving the computation.

D. Preconditions: A deciding precondition must apply to a known value.

A precondition is a predicate, used in the body of a logic procedure, that gives the criterion for choosing that procedure. The precondition may or may not be essential to the mathematical definition. But without the aid of preconditions, much backtracking may, in general, be required.

There is nothing special syntactically about the preconditions, and recognition of predicates as preconditions is totally a control issue, i.e. something known by the system that provides an interpretation for the logic program.

The following is an example of logic procedures named F in which the choice among the procedures is determined by preconditions which test the relative sizes of x and y.

$$F(x,y,z) \leftarrow (x=y) \wedge G_1(x,y,z)$$
$$F(x,y,z) \leftarrow (x>y) \wedge G_2(x,y,z)$$
$$F(x,y,z) \leftarrow (x<y) \wedge G_3(x,y,z)$$

Another example of preconditions is in the FACTOR procedure presented later. GCD

is necessary to the mathematical definition, but can also be used as both precondition and termination condition if w and n are given. In the case where p is not given, i.e. 3-invertibility, its use as a termination condition is essential to the efficiency of the algorithm.

Since the preconditions determine the efficiency of the algorithm, one must have some way of tying the preconditions to the parameters such that certain preconditions are used if certain parameters are known. A way around this problem is to have at least two preconditions, one of which can always succeed if, at most, one of the n+1 arguments of the original function are unknown. The following is an example:

$$F(x,y,z) \leftarrow (x=y) \wedge P_1(z) \wedge G_1(x,y,z)$$
$$F(x,y,z) \leftarrow (x>y) \wedge P_2(z) \wedge G_2(x,y,z)$$
$$F(x,y,z) \leftarrow (x<y) \wedge P_3(z) \wedge G_3(x,y,z)$$

Now, so long as at least two out of three of the values x, y, and z are given, either the relative sizes of x and y can be determined, or the predicate P_i can be applied to z. So, if the comparisons between x and y and the P_i's serve as preconditions, at least one will always be computable in each procedure. For the P_i's to efficiently serve as preconditions, exactly one of $P_1(z)$, $P_2(z)$, and $P_3(z)$ should be true for a given z.

4. Why is EXP1 2-invertible but not 1-invertible?

The 2-invert test on EXP1 succeeds, i.e. (101) maps to (111), but the 1-invert test fails, i.e. (011) maps to (011). Intuitively, that makes sense according to the guidelines. EXP1 fails condition A for 1-invertibility since the first parameter of the termination condition is neither constant nor can be computed from the others. The guidelines are met, however, for 2-inversion.

If we wish to achieve 1-invertibility, we can change the termination condition of EXP1. This defines a new exponentiation program which we call EXP2.

$$EXP2(1,0,1) \leftarrow$$

$$EXP2(x+1,0,z) \leftarrow EXP2(x,0,z)$$

$$EXP2(x,y+1,w) \leftarrow EXP2(x,y,z) \wedge MULT(x,z,w)$$

The computation of EXP2 is similar to that of EXP1 when the second or third parameter is the unknown.

For 1-inversion, the control is nondeterministic but will succeed. Given values for the second and third parameters, the third procedure will recur until the second parameter is reduced to zero. Application of the second procedure serves to guess a value for x; then the recursive returns test that value. An exhaustive, breadth-first search will finally discover x. However, this is not a reasonable computation. It is an example of Figure 1b. The 1-invert test on EXP2 succeeds, since (011) maps to (111), as does the 2-invert test, mapping (101) to (111). The guidelines, also, are all satisfied.

5. Another Exponentiation Algorithm.

There is an entirely different approach to exp(x,y) that is more obviously j-invertible. It is based on the fact that every positive integer, x, is a product of powers of primes, that is to say

$$x = 2^{v1}\ 3^{v2} \ldots p_k^{vk}.$$

We have $z = x^y$ iff

$$z = 2^{y \cdot v1}\ 3^{y \cdot v2} \ldots p_k^{y \cdot vk}.$$

If we are given x and y, and we can factor x, then we can construct z out of the factors and vice-versa.

We must first provide an invertible factoring predicate. Suppose we have a predicate, GCD(x,y,z), such that z is the greatest common divisor of x and y, and z

is undefined if x or y is zero. Then FACTOR(W,N,P,R) is true if and only if $W > 0$, $N > 0$, $R > 0$, $W = N^P \cdot R$, and N does not divide R. FACTOR is defined as follows:

$$\text{FACTOR}(w,n,0,w) \leftarrow \text{GCD}(w,n,1)$$

$$\text{FACTOR}(w,n,p+1,r) \leftarrow \text{GCD}(w,n,n) \wedge \text{MULT}(w',n,w) \wedge \text{FACTOR}(w',n,p,r)$$

The normal call is, for example, ← FACTOR(36,2,p,r) which yields ← FACTOR(36,2,2,9). A 1-invertible call is: FACTOR(answer,2,2,9) which yields FACTOR(36,2,2,9). These are the only ways that FACTOR will be called by EXP3.

EXP3(X,Y,Z) is true if and only if $X^Y = Z$. We have auxiliary function E such that E(X,N,Y,Z) is true if and only if $X^Y = Z$ for $X > 0$, $Z > 0$ and for all m, $2 \leq m < N$, m does not divide X.

The formal definitions of EXP3 and E are:

$$\text{EXP3}(x,y,z) \leftarrow \text{E}(x,2,y,z)$$

$$\text{E}(1,n,y,1) \leftarrow$$

$$\text{E}(x,n,y,z) \leftarrow \text{FACTOR}(x,n,p,r) \wedge \text{FACTOR}(z,n,p',r') \wedge \text{MULT}(p,y,p') \wedge \text{E}(r,n+1,y,r')$$

E repeatedly removes a prime factor, p_i, from each of x and z and checks to see that their powers are in the proper relationship, i.e. p_i^{vi} for x and $p_i^{y \cdot vi}$ for z.

The semantics of E's two procedures is:

$1^y = 1$ and no m, $2 \leq m < n$ divides 1

$(n^p \cdot r)^y = (n^{y \cdot p} \cdot r') \leftarrow (r^y = r')$ and no m, $2 \leq m < n$ divides r

EXP3 is both 1- and 2-invertible. Calling EXP3 with pattern (011) calls E with pattern (0111). The invert test for E on (0111) yields (1111). Calling EXP3 with pattern (101) calls E with pattern (1101), and the invert test on (1101) yields (1111). Notice that E passes the 3-invert test, even though the third parameter of the termination condition is neither constant nor computable from the other parameters. This demonstrates the conservatism of guideline A.

Note that E and FACTOR are not j-invertible for all j. However, that creates no difficulties. E is called with only patterns (0111), (1101), and (1110) and FACTOR with only patterns (1100) and (0111). All of those computations succeed. The invert test gives the proper answers for all cases of EXP3, E, and FACTOR, as well as all other examples in this paper.

If we wished to have a factor program that is j-invertible for all j, we could define a new one, FACTOR2.

$$\text{FACTOR2}(w,n,p,r) \leftarrow \text{EXP3}(n,p,z) \wedge \text{MULT}(z,r,w)$$

This new program is totally j-invertible, but will not produce p and r, given w and n, as FACTOR would. These characteristics of FACTOR2 are demonstrated by the mappings of the invert test on the input patterns:

$$(0111) \Rightarrow (1111)$$
$$(1011) \Rightarrow (1111)$$
$$(1101) \Rightarrow (1111)$$
$$(1110) \Rightarrow (1111)$$
$$(1100) \Rightarrow (1100)$$

6. Conclusions

We have defined the concept of j-invertibility for function and given two algorithms to test logic programs for invertibility, and while the answer is not definitive, it is indicative. The algorithms do more than just test for j-invertibility; they map

arbitrary input patterns to output patterns.

We have also presented some guidelines for constructing j-invertible functions. The guidelines are in terms of the predicate form of definition of the function, and are syntactic in nature.

The algorithm could also be used by a logic interpreter in choosing the order of evaluation of subgoals of a given procedure. That is, given a procedure invocation, including its list of arguments, the interpreter can determine a partial order on the subgoals which places the most completely evaluable subgoals first, preventing procedures from being called before they have enough information to carry out their computations. Such control can be applied dynamically by the interpreter.

Invertibility can be looked at in another way. The number of unique variables appearing in the parameters of a procedure call is the degree of freedom of that call. In general, the procedure may be able to reduce the degree of freedom by binding some of the variables. For example, MULT(x,x,9) has one degree of freedom and in fact the usual definition of MULT will yield MULT(3,3,9), computing the square root and leaving zero degrees of freedom. Similarly, MULT(x,1,z) has two degrees of freedom. MULT will determine that x = z, i.e. MULT(x,1,x), reducing the degree of freedom to one. Such situations can arise naturally in inverting predicates. How to treat them is an interesting question for which we do not yet have an answer.

REFERENCES

1. Kowalski, Robert. Predicate Logic as a Programming Language. Information Processing 74, North-Holland Publishing (1974).

2. Sickel, Sharon and McKeeman, W. M. Axiomatic Specification of Syntax-Directed Translation. Technical Report 78-8-002, Information Sciences, University of California, Santa Cruz, Ca. (1978).

3. van Emden, M. H. and Kowalski, R. A. The Semantics of Predicate Logic as a Programming Language. J.ACM 23,4 (October 1976), 733-742.

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 Copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 Copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 Copy

Office of Naval Research
Code 458
Arlington, VA 22217
1 Copy

Office of Naval Research
Branch Office, Boston
Bldg. 114, Section D
666 Summer Street
Boston, MA 02210
1 Copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, ILL 60605
1 Copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 Copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 Copies

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380
1 Copy

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 Copy

Mr. E. H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084
1 Copy

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
1 Copy